# Distributed Key Generation with Ethereum Smart Contracts

Philipp Schindler[1] ✉, Aljosha Judmayer[1],
Nicholas Stifter[1,2], and Edgar Weippl[1,2]

[1] SBA Research, Austria
{pschindler,ajudmayer,nstifter,eweippl}@sba-research.org

[2] Vienna University of Technology, Austria

**Abstract.** Distributed key generation (DKG) is a fundamental building block for a variety of cryptographic schemes and protocols, such as threshold cryptography [9], multi-party coin tossing schemes [1], public randomness beacons [20] or (BFT) consensus protocols [6,19]. More recently, the surge in interest for blockchain technologies, and in particular the quest for developing scalable protocol designs, has renewed and strengthened the need for efficient and practical DKG schemes. Surprisingly, the availability of DKG protocol implementations and analyses of their practicability is still highly limited, given their broad range of applications. We hereby help close this gap by presenting a fully functional, well documented, economically viable DKG implementation[3] for deriving keys to use with BLS threshold signatures as an Ethereum smart contract. Given the current Ethereum block gas limit ($\sim 8M$), it is possible to support up to $n = 256$ nodes while ensuring that any necessary contract call can still be executed within a single block. The practicability of our implementation is further demonstrated through the deployment and successful execution of our DKG contract in the Ropsten testnet.

## 1 Introduction

The goal of a DKG scheme is to agree on a secret that is shared among a set of $n$ participants such that only a subset of $t < n$ nodes can use or reveal the secret. Secret sharing schemes, e.g. Shamir Secret Sharing [24], or (publicly) verifiable variants, assume a (trusted) dealer that wants to share a secret it knows. In contrast, in a DKG protocol no single party has knowledge of the secret that is being shared [14].

Interestingly enough, while the topic of DKG has already been extensively discussed in the literature, for example, by the works of Genanaro et al. [12] or, more recently, Kate and Goldberg [15], practical open source implementations of DKG protocols are still rare. We aim to close this gap by providing a lightweight implementation of a DKG protocol based on modern pairing based threshold

[3] The source code, documentation, and logs of a successful execution in the Ropsten testnet are available at https://github.com/PhilippSchindler/ethdkg/.

cryptography. Our implementation consists of (i) a client application executed by each party participating in the DKG protocol, and (ii) an Ethereum smart contract which ensures that adversarial behavior of any minority of clients cannot prevent the protocol from producing the desired protocol output.

## 2 Use Cases

The ability to avoid or reduce the necessity to trust in any single third party in a DKG is an appealing characteristic, in particular in the context of permissionless cryptocurrencies and blockchain technologies. Recent improvement proposals for blockchain and other distributed ledger protocols, e.g. randomized BFT consensus protocols such as Honeybadger [17], the Dfinity blockchain protocol [13] or Calypso [16], are increasingly relying on threshold cryptography as part of the presented scheme. In this scenarios, one typically does not want to trust a single centralized entity for the required protocol setup, but instead use DKG protocols as an alternative without trusted authorities. Our DKG protocol is particularly suitable for such scenarios as it can be executed on the readily available Ethereum platform, provides flexible means for registration of the participants, and tolerates faulty or adversarial behavior of any minority of participants.

Our smart contract based DKG protocol can also be used to bootstrap a variety of interesting applications on the Ethereum platform itself. A candidate example is a decentralized source of publicly-verifiable, bias-resistant and unpredictable randomness, or short, a randomness beacon [20]. By levering the security and uniqueness properties of BLS threshold signatures, the construction of a randomness beacon follows naturally. Within the Ethereum platform, such trustworthy randomness beacons are particularly useful, as there are currently no built-in mechanisms for deriving randomness with the aforementioned characteristics. Consequently, Ethereum smart contracts largely rely on less secure sources of randomness (such as the block hash) or even depend on trusted third parties. Beyond an application in Ethereum smart contracts, unpredictable bias-resistant public randomness also plays an important role in a broad range of fields, including proof-of-stake and sharding protocols, privacy preserving messaging services, e-voting protocols, as well as gambling and lottery services [23].

Another Ethereum related use case for our DKG includes wallet contracts where multiple signatures are required to initiate some action, such as performing transactions, which can be verified with a constant amount of computation in the contract.

Our DKG can also be used to setup the public key in a threshold public key encryption scheme [8]. In such a scenario, a client could encrypt a message under the generated master public key such that the decryption of the message requires the collaboration of a threshold of participants from the DKG. Combined with smart contracts, one can construct an incentive-based timelock encryption protocol [22].

## 3 Related Work

To the best of our knowledge, the DKG protocol [18] developed by the Orbs Network team is the only publicly available protocol targeting a similar deployment scenario, namely, an implementation of a DKG protocol using the Ethereum platform. However, the presented prototypical implementation appears to be incomplete and has not been updated in over 4 months. For example, countermeasures to protect against key cancellation attacks [2] are not (yet) implemented, although documentation in the contract source code (`dkg.sol`) suggests that the team is aware of the issue. In the herein presented DKG protocol we provide a mechanism to protect against these kinds of attacks using a non-interactive zero-knowledge proof, which can be efficiently verified by the smart contract. Additionally, our protocol can tolerate up to $f = \lceil \frac{n}{2} \rceil - 1$ Byzantine participants while still completing successfully. In contrast, the Orbs Network implementation requires a protocol restart even if just a single adversarial participant sends an invalid share. Furthermore, we not only provide an implementation of the smart contract itself, but also make the the implementation of the client software that handles the necessary interactions with the Ethereum blockchain publicly available.

## 4 High Level Protocol Description

Our DKG protocol is tailored towards a practical implementation to be used with the Ethereum blockchain and was not yet presented in a standalone setting. The protocol is inspired by the original presentation of secret sharing by Shamir [24] and the aggregation properties of the BLS signature scheme [2,3,4]. Our protocol operates in four consecutive phases and the smart contract ensures that the functions can only be executed during the correct phases. Further, the smart contract and client software upholds sufficient waiting times between phases, to ensure that agreement on a common-prefix [11] of the Ethereum blockchain is reached (with high probability) before taking state dependent actions. The protocol phases are:

1. **Registration:** Each participant submits a transaction to the smart contract that contains an individual public key.
2. **Key Sharing:** All Participants perform secret sharing of their private keys with the nodes successfully registered in the previous step.
3. **Dispute:** Participants may complain about invalid shares received.
4. **Finalization:** The master public key is constructed, uploaded and verified.

In the following, we discuss each of the protocol phases in more detail.

### 4.1 Registration

During the registration phase, each participant prepares a BLS keypair and sends the public key together with a non-interactive zero-knowledge (NIZK) proof of knowledge [7] of the secret key to the smart contract. This proof is required to protect against rogue key attacks [2], where an adversary crafts a public key in

a way that, upon aggregation, cancels out the keys from other participants. Furthermore we use this proof to tie the participant's externally owned Ethereum account to the BLS public key submitted, to prevent an adversary from registering themselves with a public key copied from an honest participant. The contract verifies the proof, stores the public key received, assigns the registrant an incrementing id $1, 2, ..., n$ and triggers a Registration event to notify all participants.

### 4.2 Key Sharing

Each of the $n$ registered participants uses Shamir's Secret Sharing [24] to share its BLS secret key among all previously registered nodes. We set the secret sharing threshold $t = \lfloor \frac{n}{2} \rfloor + 1$. The shares are encrypted using a shared key, derived from the issuer's secret key and the receiver's registered public key, similar to the Diffie-Hellman key exchange protocol [10]. This construction ensures an that all shared keys (between two correct clients) are different without requiring to verifiably distribute $\mathcal{O}(n^2)$ keys. In section 4.3, we show how to use this property to deal with disputes without having to abort the protocol under adversarial behavior. To allow for share verification, each participant further needs to commit to the coefficients of the secret sharing polynomial. The encrypted shares and commitment are sent as a transaction to the smart contract, which triggers a KeySharing event upon successful processing in the smart contract. Upon receiving a transaction with encrypted shares, the smart contract verifies that (i) a participant is eligible to upload (i.e. that the contract is in the key sharing phase and the participants has previously registered) and (ii) has provided the correct number of shares $(n - 1)$ and commitments $(t)$. However the contract does not (and cannot) decrypt the provided shares and check their validity. This verification is performed by the client software, which is actively monitoring the p2p network for newly mined blocks and is listening for KeySharing events. When such an event is fired, the client first decrypts their respective share and then verifies the decrypted share against the commitment to ensure its correctness.

### 4.3 Dispute

If a participant discovers that one (or more) of its received shares are invalid, it files a dispute for each of the individual shares. For this purpose, the decryption key (i.e. the shared key between the malicious share issuer and itself) is sent as part of the dispute transaction to the smart contract. The smart contract can then attempt to decrypt the share and check for its correctness. However, the contract additionally needs to verify that the provided decryption key is indeed correct. Otherwise an adversary could easily claim that shares it received are invalid by providing an invalid decryption key. To allow for this verification, in addition to the decryption key, a participant has to provide a NIZK proof [7] showing that the decryption key is indeed valid. Notice that sending a dispute transaction renders the corresponding decryption key public knowledge. However, as we ensure that all decryption keys (between correct nodes) are different, the adversary cannot learn any additional information by forcing a correct node to submit a dispute.

### 4.4 Finalization

During the finalization phase each participant first determines the set of participants that successfully shared their secret key. A participant is only part of this set if it registered and shared its key successfully and there were no successful disputes filed against the participant. After this set is established, each participant can compute its (individual) group secret key by computing the sum of the received shares. Furthermore the master public key can be computed by adding all public keys. As soon as any of the participants uploads this master public key, and the smart contract verifies its correctness, threshold signatures under this public key can be verified by the smart contract.

## 5 Evaluation

To show the viability of our protocol we deployed our DKG contract in the Ethereum test network Ropsten and simulated a simple scenario with 5 participants running the client software. Clients A, B and C follow the prescribed protocol, while client D aborts after the registration phase and client E actively tries to manipulate the protocol run by providing an invalid share. The contract can be found using an Ethereum block explorer such as `https://ropsten.etherscan.io/` by supplying the contract address `0x64eB9cbc8AAc7723A7A94b178b7Ac4c18D7E6269`. This exemplary protocol execution is further documented in the Github repository `https://github.com/PhilippSchindler/ethdkg/evaluation/`.

To highlight the scalability of our approach, we provide gas usage measurements for all types of transactions a client (potentially) needs to execute during a protocol run. We tested our implementation with up to 256 nodes and observe that the consumed gas for all transactions is well below the current block gas limit of $\sim 8M$. Deployment of a contract instance consumed $\sim 3.5M$ gas. The most expensive operation, i.e. filing a dispute, only has to be executed if an adversary actively tries to manipulate the protocol run. As in this case the adversarial behavior can be cryptographically proven to the smart contract, a security deposit could be used to refund the costs for a rightful dispute claim. Using the suggested gas price estimate of 2.5 Gwei from `https://ethgasstation.info/`, the cost of submitting a dispute in a scenario with 256 nodes is only $\sim 0.018$ ETH ($\sim$ \$1.50).

## 6 Challenges

One of the major challenges faced is the implementation of the required cryptographic primitives within the constraints imposed by the Ethereum platform. While elliptic curve operations such as additions, multiplications and pairings are efficiently computable on modern hardware, one needs to use a very limited set of instructions (i.e. precompiled contracts [21,5]) to efficiently perform these calculations within the Ethereum EVM. In principle, one could implement any required operation using the available EVM opcodes. In practice, this however leads to very high gas consumption for more involved computations.

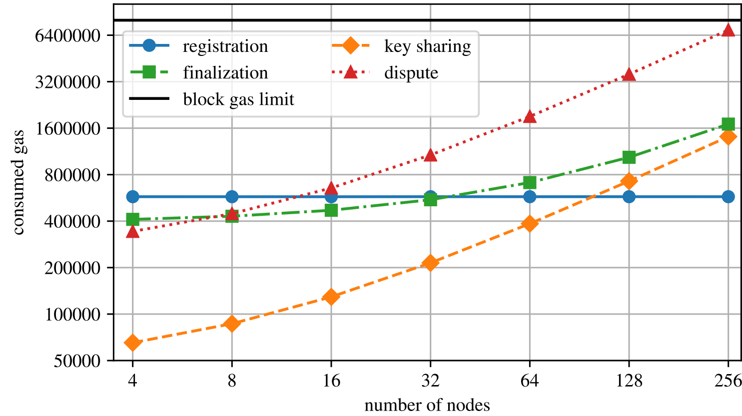


**Fig. 1.** Measured gas consumption per contract phase and participant

Consequently, it is necessary to stick to the built-in primitives, and find efficient alternatives for operations which are not available directly. One particular limitation we had to work around is that there are no elliptic curve additions and multiplications for one of the two groups used for BLS signatures. A possible solution used to overcome this problem is to perform the computations offline and verify the correctness using the precompiled pairing contract. The lack of built-in operations for e.g. symmetric key encryption or signature verification further necessitated the implementation of practical alternatives by hand. To give another example for the faced challenges, we recall the BLS signature verification mechanism: a signature $\sigma$ is valid if and only if the pairing check $e(\sigma, g_2) = e(H(m), pubkey)$ is successful. The pairing check itself can be performed efficiently using a precompiled contract in Ethereum. The involved hash function $H(\cdot)$ however needs to map into an elliptic curve group. Consequently, the built-in primitives for Keccak or SHA-256 hash functions cannot be used directly. We follow the description of Boneh et al. [4] to implement a proper hash function mapping map into the elliptic curve group.

Considering the protocol design itself, we present an *efficient* solution for following challenges (in increasing complexity): (i) detection of malicious actions, (ii) cryptographically proving such behavior within the smart contract, and (iii) tolerating any minority of Byzantine participants without a protocol restart.

## 7 Conclusion

In this paper we highlight the importance of practical DKG protocols for a wide range of cryptographic schemes and applications. We provide a working and documented open source implementation of a practical DKG protocol ready for deployment on the Ethereum platform. Thereby, we show how major environmental challenges can be overcome. Despite the constraints imposed by Ethereum and the EVM, we not only demonstrate the feasibly of our approach, but also highlight its practicability in regard to gas costs and the possible number of participants.

## References


1. Blum, M.: Coin flipping by telephone a protocol for solving impossible problems. ACM SIGACT News **15**(1), 23–27 (1983)
2. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 435–464. Springer (2018)
3. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 416–432. Springer (2003)
4. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 514–532. Springer (2001)
5. Buterin, V., Reitwiessner, C.: EIP 197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128 (2018), https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md, Accessed: 2018-12-15
6. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. pp. 123–132. ACM (2000), https://www.zurich.ibm.com/~cca/papers/abba.pdf
7. Camenisch, J., Stadler, M.: Proof systems for general statements about discrete logarithms. Technical report/Dept. of Computer Science, ETH Zürich **260** (1997)
8. Cramer, R., Gennaro, R., Schoenmakers, B.: A secure and optimally efficient multi-authority election scheme. European transactions on Telecommunications **8**(5), 481–490 (1997)
9. Desmedt, Y., Frankel, Y.: Threshold cryptosystems. In: Brassard, G. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 435, pp. 307–315. Springer (1989)
10. Diffie, W., Hellman, M.: New directions in cryptography. IEEE transactions on Information Theory **22**(6), 644–654 (1976)
11. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 281–310. Springer (2015)
12. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 295–310. Springer (1999)
13. Hanke, T., Movahedi, M., Williams, D.: Dfinity technology overview series consensus system (2018), https://dfinity.org/pdf-viewer/library/dfinity-consensus.pdf, rev. 1
14. Kate, A.: Distributed key generation and its applications (2010)
15. Kate, A., Goldberg, I.: Distributed key generation for the internet. In: 2009 29th IEEE International Conference on Distributed Computing Systems. pp. 119–128. IEEE (2009)
16. Kokoris-Kogias, E., Alp, E.C., Siby, S.D., Gailly, N., Gasser, L., Jovanovic, P., Syta, E., Ford, B.: Calypso: Auditable sharing of private data over blockchains. Cryptology ePrint Archive, Report 2018/209 (2018), https://eprint.iacr.org/2018/209
17. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of bft protocols. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 31–42. ACM (2016)

18. Orbs Network: DKG for BLS threshold signature scheme on the EVM using solidity (2018), https://github.com/orbs-network/dkg-on-evm, Accessed: 2018-12-11
19. Rabin, M.O.: Randomized byzantine generals. In: Foundations of Computer Science, 1983., 24th Annual Symposium on. pp. 403–409. IEEE (1983), https://www.cs.princeton.edu/courses/archive/fall05/cos521/byzantin.pdf
20. Rabin, M.O.: Transaction protection by beacons. Journal of Computer and System Sciences **27**(2), 256–267 (1983)
21. Reitwiessner, C.: EIP 196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128 (2018), https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md, Accessed: 2018-12-15
22. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
23. Schindler, P., Judmayer, A., Stifter, N., Weippl, E.: Hydrand: Practical continuous distributed randomness. IACR Cryptology ePrint Archive **2018**, 319 (2018)
24. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)